下载APP

06｜构建模式：Go是怎么解决包依赖管理问题的？

2021-10-25 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 27:24 大小 25.11M

你好，我是 Tony Bai。

通过前面的讲解，我们已经初步了解了 Go 程序的结构，以及 Go 项目的典型布局了。那么，接下来，**我们是时候来系统学习一下 Go 应用的构建了**，它们都是我们继续 Go 语言学习的前提。

所以在这一节课，我们就来了解 Go 构建模式演化的前世今生。理解了这个发展史后，我们会重点来探讨现在被广泛采用的构建模式，Go Module 的基本概念和应用构建方式。接着，知道了怎么做后，我们会再深一层，继续分析 Go Module 的工作原理。这样层层深入地分析完后，你就能彻底、透彻地掌握 Go Module 构建模式了。

好了，我们直接开始吧。我们先来了解一下 Go 构建模式的演化过程，弄清楚 Go 核心开发团队为什么要引入 Go module 构建模式。

Go 构建模式是怎么演化的？

Go 程序由 Go 包组合而成的，**Go 程序的构建过程就是确定包版本、编译包以及将编译后得到的目标文件链接在一起的过程。**

Go 语言的构建模式历经了三个迭代和演化过程，分别是最初期的 GOPATH、1.5 版本的 Vendor 机制，以及现在的 Go Module。这里我们就先来介绍一下前面这两个。

首先我们来看 GOPATH。

Go 语言在首次开源时，就内置了一种名为 **GOPATH** 的构建模式。在这种构建模式下，Go 编译器可以在本地 GOPATH 环境变量配置的路径下，搜寻 Go 程序依赖的第三方包。如果存在，就使用这个本地包进行编译；如果不存在，就会报编译错误。

我这里给出了一段在 GOPATH 构建模式下编写的代码，你先来感受一下：

复制代码

```
1 package main
2
3 import "github.com/sirupsen/logrus"
4
5 func main() {
6     logrus.Println("hello, gopath mode")
7 }
```

你可以看到，这段代码依赖了第三方包 logrus（logrus 是 Go 社区使用最为广泛的第三方 log 包）。

接下来，这个构建过程演示了 Go 编译器（这里使用 Go 1.10.8）在 GOPATH 环境变量所配置的目录下（这里为 /Users/tonybai/Go），无法找到程序依赖的 logrus 包而报错的情况：

复制代码

```
1 $go build main.go
```

```
main.go:3:8: cannot find package "github.com/sirupsen/logrus" in any of:
  /Users/tonybai/.bin/go1.10.8/src/github.com/sirupsen/logrus (from $GOROOT)
  /Users/tonybai/Go/src/github.com/sirupsen/logrus (from $GOPATH)
```

那么 Go 编译器在 GOPATH 构建模式下，究竟怎么在 GOPATH 配置的路径下搜寻第三方依赖包呢？

为了给你说清楚搜寻规则，我们先假定 Go 程序导入了 github.com/user/repo 这个包，我们也同时假定当前 GOPATH 环境变量配置的值为：

复制代码

```
export GOPATH=/usr/local/goprojects:/home/tonybai/go
```

那么在 GOPATH 构建模式下，Go 编译器在编译 Go 程序时，就会在下面两个路径下搜索第三方依赖包是否存在：

复制代码

```
/usr/local/goprojects/src/github.com/user/repo
/home/tonybai/go/src/github.com/user/repo
```

这里注意一下，如果你没有显式设置 GOPATH 环境变量，Go 会将 GOPATH 设置为默认值，不同操作系统下默认值的路径不同，在 macOS 或 Linux 上，它的默认值是 $HOME/go。

那么，当遇到像上面例子一样，没有在本地找到程序的第三方依赖包的情况，我们该如何解决这个问题呢？

这个时候就要让 go get 登场了！

我们可以通过 go get 命令将本地缺失的第三方依赖包下载到本地，比如：

复制代码

```
$go get github.com/sirupsen/logrus
```

这里的 go get 命令，不仅能将 logrus 包下载到 GOPATH 环境变量配置的目录下，它还会检查 logrus 的依赖包在本地是否存在，如果不存在，go get 也会一并将它们下载到本地。

不过，go get 下载的包只是那个时刻各个依赖包的最新主线版本，这样会给后续 Go 程序的构建带来一些问题。比如，依赖包持续演进，可能会导致不同开发者在不同时间获取和编译同一个 Go 包时，得到不同的结果，也就是不能保证可重现的构建（Reproduceable Build）。又比如，如果依赖包引入了不兼容代码，程序将无法通过编译。

最后还有一点，如果依赖包因引入新代码而无法正常通过编译，并且该依赖包的作者又没用及时修复这个问题，这种错误也会传导到你的程序，导致你的程序无法通过编译。

也就是说，**在 GOPATH 构建模式下，Go 编译器实质上并没有关注 Go 项目所依赖的第三方包的版本。**但 Go 开发者希望自己的 Go 项目所依赖的第三方包版本能受到自己的控制，而不是随意变化。于是 Go 核心开发团队引入了 Vendor 机制试图解决上面的问题。

现在我们就来看看 vendor 机制是怎么解决这个问题的。

Go 在 1.5 版本中引入 **vendor 机制**。vendor 机制本质上就是在 Go 项目的某个特定目录下，将项目的所有依赖包缓存起来，这个特定目录名就是 vendor。

Go 编译器会优先感知和使用 vendor 目录下缓存的第三方包版本，而不是 GOPATH 环境变量所配置的路径下的第三方包版本。这样，无论第三方依赖包自己如何变化，无论 GOPATH 环境变量所配置的路径下的第三方包是否存在、版本是什么，都不会影响到 Go 程序的构建。

如果你将 vendor 目录和项目源码一样提交到代码仓库，那么其他开发者下载你的项目后，就可以实现可重现的构建。因此，如果使用 vendor 机制管理第三方依赖包，最佳实践就是将 vendor 一并提交到代码仓库中。

下面这个目录结构就是为上面的代码示例添加 vendor 目录后的结果：

复制代码

```
1 .
2 ├── main.go
```

```
└── vendor/
    ├── github.com/
    │   └── sirupsen/
    │       └── logrus/
    └── golang.org/
        └── x/
            └── sys/
                └── unix/
```

在添加完 vendor 后，我们重新编译 main.go，这个时候 Go 编译器就会在 vendor 目录下搜索程序依赖的 logrus 包以及后者依赖的 golang.org/x/sys/unix 包了。

这里你还要注意一点，要想开启 vendor 机制，你的 Go 项目必须位于 GOPATH 环境变量配置的某个路径的 src 目录下面。如果不满足这一路径要求，那么 Go 编译器是不会理会 Go 项目目录下的 vendor 目录的。

不过 vendor 机制虽然一定程度解决了 Go 程序可重现构建的问题，但对开发者来说，它的体验却不那么好。一方面，Go 项目必须放在 GOPATH 环境变量配置的路径下，庞大的 vendor 目录需要提交到代码仓库，不仅占用代码仓库空间，减慢仓库下载和更新的速度，而且还会干扰代码评审，对实施代码统计等开发者效能工具也有比较大影响。

另外，你还需要手工管理 vendor 下面的 Go 依赖包，包括项目依赖包的分析、版本的记录、依赖包获取和存放，等等，**最让开发者头疼的就是这一点。**

为了解决这个问题，Go 核心团队与社区**将 Go 构建的重点转移到如何解决包依赖管理上**。Go 社区先后开发了诸如 gb、glide、dep 等工具，来帮助 Go 开发者对 vendor 下的第三方包进行自动依赖分析和管理，但这些工具也都有自身的问题。

就在 Go 社区为包依赖管理焦虑并抱怨没有官方工具的时候，Go 核心团队基于社区实践的经验和教训，推出了 Go 官方的解决方案：**Go Module**。

创建你的第一个 Go Module

从 Go 1.11 版本开始，除了 GOPATH 构建模式外，Go 又增加了一种 Go Module 构建模式。

在04 讲中，我们曾基于 Go Module 构建模式编写过一个“hello, world”程序，当时是为了讲解 Go 程序结构，这里我再带你回顾一下 Go Module 的基础概念。

一个 Go Module 是一个 Go 包的集合。module 是有版本的，所以 module 下的包也就有了版本属性。这个 module 与这些包会组成一个独立的版本单元，它们一起打版本、发布和分发。

在 Go Module 模式下，通常一个代码仓库对应一个 Go Module。一个 Go Module 的顶层目录下会放置一个 go.mod 文件，每个 go.mod 文件会定义唯一一个 module，也就是说 Go Module 与 go.mod 是一一对应的。

go.mod 文件所在的顶层目录也被称为 module 的根目录，module 根目录以及它子目录下的所有 Go 包均归属于这个 Go Module，这个 module 也被称为 main module。

你可能也意识到了，Go Module 的原理和使用方法其实有点复杂，但“千里之行始于足下”，下面我们先从如何创建一个 Go Module 说起。我们先来将上面的例子改造成为一个基于 Go Module 构建模式的 Go 项目。

创建一个 Go Module

将基于当前项目创建一个 Go Module，通常有如下几个步骤：

第一步，通过 go mod init 创建 go.mod 文件，将当前项目变为一个 Go Module；

第二步，通过 go mod tidy 命令自动更新当前 module 的依赖信息；

第三步，执行 go build，执行新 module 的构建。

我们一步一步来详细看一下。

我们先建立一个新项目 module-mode 用来演示 Go Module 的创建，注意我们可以在任意路径下创建这个项目，不必非要在 GOPATH 环境变量的配置路径下。

这个项目的 main.go 修改自上面的例子，修改后的 main.go 的代码是这样的，我们依旧依赖外部包 logrus：

复制代码

```
package main

import "github.com/sirupsen/logrus"

func main() {
  logrus.Println("hello, go module mode")
}
```

你可以看到，这个项目目录下只有 main.go 一个源文件，现在我们就来为这个项目添加 Go Module 支持。我们通过 go mod init 命令为这个项目创建一个 Go Module（这里我们使用的是 Go 版本为 1.16.5，Go 1.16 版本默认采用 Go Module 构建模式）：

复制代码

```
$go mod init github.com/bigwhite/module-mode
go: creating new go.mod: module github.com/bigwhite/module-mode
go: to add module requirements and sums:
  go mod tidy
```

现在，go mod init 在当前项目目录下创建了一个 go.mod 文件，这个 go.mod 文件将当前项目变为了一个 Go Module，项目根目录变成了 module 根目录。go.mod 的内容是这样的：

复制代码

```
module github.com/bigwhite/module-mode

go 1.16
```

这个 go.mod 文件现在处于初始状态，它的第一行内容用于声明 module 路径 (module path)，最后一行是一个 Go 版本指示符，用于表示这个 module 是在某个特定的 Go 版本的 module 语义的基础上编写的。

go mod init 命令还输出了两行日志，提示我们可以使用 go mod tidy 命令，添加 module 依赖以及校验和。go mod tidy 命令会扫描 Go 源码，并自动找出项目依赖的外部 Go Module 以及版本，下载这些依赖并更新本地的 go.mod 文件。我们按照这个提示执行一下 go mod tidy 命令：

复制代码

```
$go mod tidy
go: finding module for package github.com/sirupsen/logrus
go: downloading github.com/sirupsen/logrus v1.8.1
go: found github.com/sirupsen/logrus in github.com/sirupsen/logrus v1.8.1
go: downloading golang.org/x/sys v0.0.0-20191026070338-33540a1f6037
go: downloading github.com/stretchr/testify v1.2.2
```

我们看到，对于一个处于初始状态的 module 而言，go mod tidy 分析了当前 main module 的所有源文件，找出了当前 main module 的所有第三方依赖，确定第三方依赖的版本，还下载了当前 main module 的直接依赖包（比如 logrus），以及相关间接依赖包（直接依赖包的依赖，比如上面的 golang.org/x/sys 等）。

Go Module 还支持通过 Go Module 代理服务加速第三方依赖的下载。在 03 讲我们讲解 Go 环境安装时，就提到过 GOPROXY 环境变量，这个环境变量的默认值为“**https://proxy.golang.org,direct**”，不过我们可以配置更适合于中国大陆地区的 Go Module 代理服务。

由 go mod tidy 下载的依赖 module 会被放置在本地的 module 缓存路径下，默认值为 $GOPATH[0]/pkg/mod，Go 1.15 及以后版本可以通过 GOMODCACHE 环境变量，自定义本地 module 的缓存路径。

执行 go mod tidy 后，我们示例 go.mod 的内容更新如下：

复制代码

```
module github.com/bigwhite/module-mode

go 1.16

require github.com/sirupsen/logrus v1.8.1
```

你可以看到，当前 module 的直接依赖 logrus，还有它的版本信息都被写到了 go.mod 文件的 require 段中。

而且，执行完 go mod tidy 后，当前项目除了 go.mod 文件外，还多了一个新文件 go.sum，内容是这样的：

复制代码

```
github.com/davecgh/go-spew v1.1.1 h1:vj9j/u1bqnvCEfJOwUhtlOARqs3+rkHYY13jYWTU9
github.com/davecgh/go-spew v1.1.1/go.mod h1:J7Y8YcW2NihsgmVo/mv3lAwl/skON4iLHj
github.com/pmezard/go-difflib v1.0.0 h1:4DBwDE0NGyQoBHbLQYPwSUPoCMWR5BEzIk/f1l
github.com/pmezard/go-difflib v1.0.0/go.mod h1:iKH77koFhYxTK1pcRnkKkqfTogsbg7g
github.com/sirupsen/logrus v1.8.1 h1:dJKuHgqk1NNQlqoA6BTlM1Wf9DOH3NBjQyu0h9+AZ
github.com/sirupsen/logrus v1.8.1/go.mod h1:yWOB1SBYBC5VeMP7gHvWumXLIWorT60ONW
github.com/stretchr/testify v1.2.2 h1:bSDNvY7ZPG5RlJ8otE/7V6gMiyenm9RtJ7IUVIAo
github.com/stretchr/testify v1.2.2/go.mod h1:a8OnRcib4nhh0OaRAV+Yts87kKdq0PP7p
golang.org/x/sys v0.0.0-20191026070338-33540a1f6037 h1:YyJpGZS1sBuBCzLAR1VEpK1
golang.org/x/sys v0.0.0-20191026070338-33540a1f6037/go.mod h1:h1NjWce9XRLGQEsW
```

这同样是由 go mod 相关命令维护的一个文件，它存放了特定版本 module 内容的哈希值。

这是 Go Module 的一个安全措施。当将来这里的某个 module 的特定版本被再次下载的时候，go 命令会使用 go.sum 文件中对应的哈希值，和新下载的内容的哈希值进行比对，只有哈希值比对一致才是合法的，这样可以确保你的项目所依赖的 module 内容，不会被恶意或意外篡改。因此，我推荐你把 go.mod 和 go.sum 两个文件与源码，一并提交到代码版本控制服务器上。

现在，go mod init 和 go mod tidy 已经为我们当前 Go Module 的构建铺平了道路，接下来，**我们只需在当前 module 的根路径下，执行 go build 就可以完成 module 的构建了**！

go build 命令会读取 go.mod 中的依赖及版本信息，并在本地 module 缓存路径下找到对应版本的依赖 module，执行编译和链接。如果顺利的话，我们会在当前目录下看到一个新生成的可执行文件 module-mode，执行这个文件我们就能得到正确结果了。

整个过程的执行步骤是这样的：

复制代码

```
$go build
$$ls
go.mod    go.sum    main.go    module-mode*
$./module-mode
INFO[0000] hello, go module mode
```

好了，到这里，我们已经完成了一个有着多个第三方依赖的项目的构建了。但关于 Go Module 的操作还远不止这些。随着 Go 项目的演进，我们会在代码中导入新的第三方包，删除一些旧的依赖包，更新一些依赖包的版本等。关于这些内容，我会在下一节课再给你详细讲解。

那么，在看到我们的 Go Module 机制会自动分析项目的依赖包，并选出最适合的版本后，不知道你会不会有这样的疑惑：**项目所依赖的包有很多版本，Go Module 是如何选出最适合的那个版本的呢？**要想回答这个问题，我们就需要深入到 Go Module 构建模式的工作原理中去。

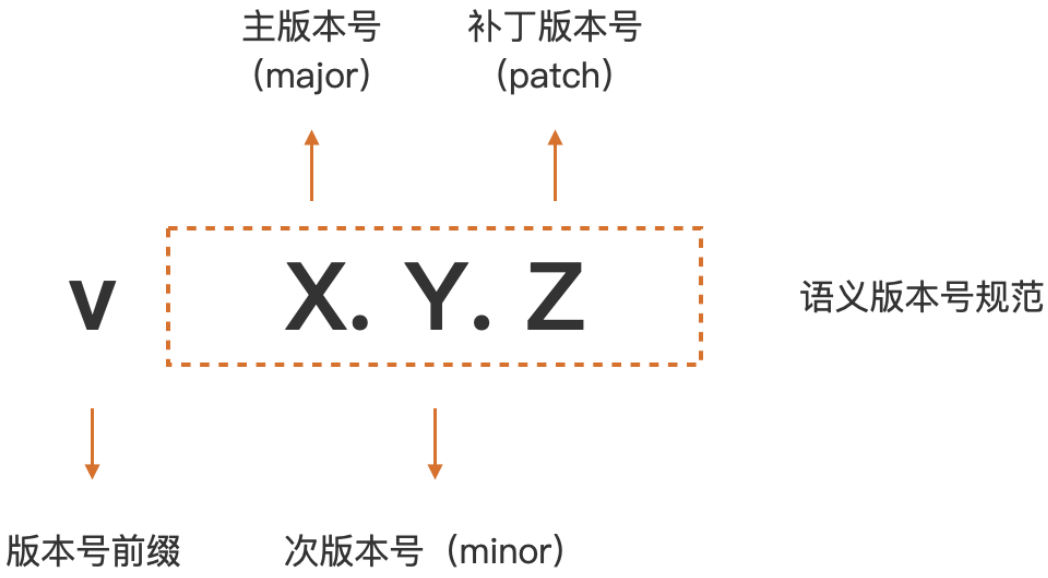
深入 Go Module 构建模式

Go 语言设计者在设计 Go Module 构建模式，来解决“包依赖管理”的问题时，进行了几项创新，这其中就包括**语义导入版本 (Semantic Import Versioning)**，以及和其他主流语言不同的**最小版本选择 (Minimal Version Selection)** 等机制。只要你深入理解了这些机制，你就能真正掌握 Go Module 构建模式。

首先我们看一下 Go Module 的语义导入版本机制。

在上面的例子中，我们看到 go.mod 的 require 段中依赖的版本号，都符合 **vX.Y.Z** 的格式。在 Go Module 构建模式下，一个符合 Go Module 要求的版本号，由前缀 v 和一个满足语义版本规范的版本号组成。

你可以看看下面这张图，语义版本号分成 3 部分：主版本号 (major)、次版本号 (minor) 和补丁版本号 (patch)。例如上面的 logrus module 的版本号是 v1.8.1，这就表示它的主版本号为 1，次版本号为 8，补丁版本号为 1。



Go 命令和 go.mod 文件都使用上面这种符合语义版本规范的版本号，作为描述 Go Module 版本的标准形式。借助于语义版本规范，Go 命令可以确定同一 module 的两个版本发布的先后次序，而且可以确定它们是否兼容。

按照语义版本规范，主版本号不同的两个版本是相互不兼容的。而且，在主版本号相同的情况下，次版本号大都是向后兼容次版本号小的版本。补丁版本号也不影响兼容性。

而且，Go Module 规定：**如果同一个包的新旧版本是兼容的，那么它们的包导入路径应该是相同的**。怎么理解呢？我们来举个简单示例。我们就以 logrus 为例，它有很多发布版本，我们从中选出两个版本 v1.7.0 和 v1.8.1.。按照上面的语义版本规则，这两个版本的主版本号相同，新版本 v1.8.1 是兼容老版本 v1.7.0 的。那么，我们就可以知道，如果一个项目依赖 logrus，无论它使用的是 v1.7.0 版本还是 v1.8.1 版本，它都可以使用下面的包导入语句导入 logrus 包：

复制代码

```
1 import "github.com/sirupsen/logrus"
```

那么问题又来了，假如在未来的某一天，logrus 的作者发布了 logrus v2.0.0 版本。那么根据语义版本规则，该版本的主版本号为 2，已经与 v1.7.0、v1.8.1 的主版本号不同了，那么 v2.0.0 与 v1.7.0、v1.8.1 就是不兼容的包版本。然后我们再按照 Go Module 的规

定，如果一个项目依赖 logrus v2.0.0 版本，那么它的包导入路径就不能再与上面的导入方式相同了。那我们应该使用什么方式导入 logrus v2.0.0 版本呢？

Go Module 创新性地给出了一个方法：将包主版本号引入到包导入路径中，我们可以像下面这样导入 logrus v2.0.0 版本依赖包：

复制代码

```
import "github.com/sirupsen/logrus/v2"
```

这就是 Go 的“语义导入版本”机制，也就是说通过在包导入路径中引入主版本号的方式，来区别同一个包的不兼容版本，这样一来我们甚至可以同时依赖一个包的两个不兼容版本：

复制代码

```
import (
    "github.com/sirupsen/logrus"
    logv2 "github.com/sirupsen/logrus/v2"
)
```

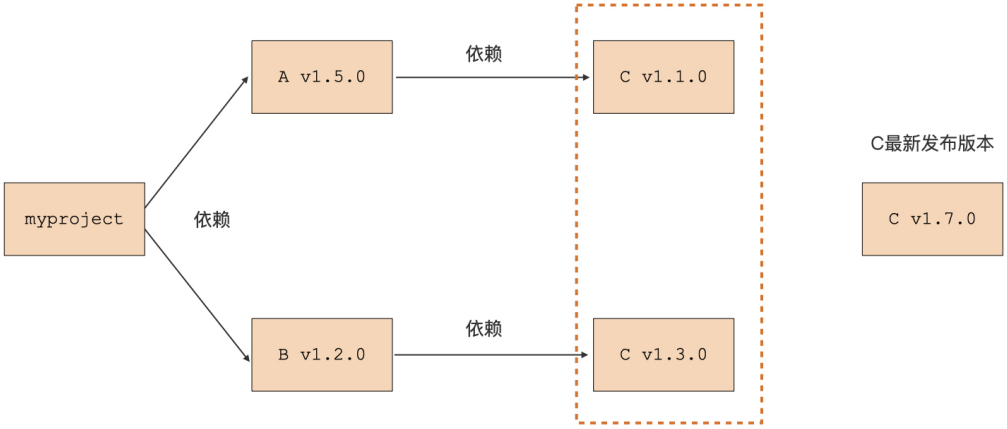
不过到这里，你可能会问，v0.y.z 版本应该使用哪种导入路径呢？

按照语义版本规范的说法，v0.y.z 这样的版本号是用于项目初始开发阶段的版本号。在这个阶段任何事情都有可能发生，其 API 也不应该被认为是稳定的。Go Module 将这样的版本 (v0) 与主版本号 v1 做同等对待，也就是采用不带主版本号的包导入路径，这样一定程度降低了 Go 开发人员使用这样版本号包时的心智负担。

Go 语义导入版本机制是 Go Module 机制的基础规则，同样它也是 Go Module 其他规则的基础。

接下来，我们再来看一下 Go Module 的最小版本选择原则。

在前面的例子中，Go 命令都是在项目初始状态分析项目的依赖，并且项目中两个依赖包之间没有共同的依赖，这样的包依赖关系解决起来还是比较容易的。但依赖关系一旦复杂起来，比如像下图中展示的这样，Go 又是如何确定使用依赖包 C 的哪个版本的呢？



在这张图中，myproject 有两个直接依赖 A 和 B，A 和 B 有一个共同的依赖包 C，但 A 依赖 C 的 v1.1.0 版本，而 B 依赖的是 C 的 v1.3.0 版本，并且此时 C 包的最新发布版为 C v1.7.0。这个时候，Go 命令是如何为 myproject 选出间接依赖包 C 的版本呢？选出的究竟是 v1.7.0、v1.1.0 还是 v1.3.0 呢？你可以暂停一两分钟思考一下。

其实，当前存在的主流编程语言，以及 Go Module 出现之前的很多 Go 包依赖管理工具都会**选择依赖项的“最新最大 (Latest Greatest) 版本”**，对应到图中的例子，这个版本就是 v1.7.0。

当然了，理想状态下，如果语义版本控制被正确应用，并且这种“社会契约”也得到了很好的遵守，那么这种选择算法是有道理的，而且也可以正常工作。在这样的情况下，依赖项的“最新最大版本”应该是最稳定和安全的版本，并且应该有向后兼容性。至少在相同的主版本 (Major Verion) 依赖树中是这样的。

但我们这个问题的答案并不是这样的。Go 设计者另辟蹊径，在诸多兼容性版本间，他们不光要考虑最新最大的稳定与安全，还要尊重各个 module 的述求：A 明明说只要求 C v1.1.0，B 明明说只要求 C v1.3.0。**所以 Go 会在该项目依赖项的所有版本中，选出符合项目整体要求的“最小版本”。**

这个例子中，C v1.3.0 是符合项目整体要求的版本集合中的版本最小的那个，于是 Go 命令选择了 C v1.3.0，而不是最新最大的 C v1.7.0。并且，Go 团队认为“最小版本选择”为

Go 程序实现持久的和可重现的构建提供了最佳的方案。

了解了语义导入版本与最小版本选择两种机制后，你就可以说你已经掌握了 Go Module 的精髓。

但很多 Go 开发人员的起点，并非是默认开启 Go Module 构建模式的 Go 1.16 版本，多数 Go 开发人使用的环境中都存在着多套 Go 版本，有用于体验最新功能特性的 Go 版本，也有某些遗留项目所使用的老版本 Go 编译器。

它们工作时采用的构建模式是不一样的，并且即便是引入 Go Module 的 Go 1.11 版本，它的 Go Module 机制，和后续进化后的 Go 版本的 Go Module 构建机制在表现行为上也有所不同。因此 Go 开发人员可能需要经常在各个 Go 版本间切换。而明确具体版本下 Go Module 的实际表现行为对 Go 开发人员是十分必要的。

Go 各版本构建模式机制和切换

我们前面说了，在 Go 1.11 版本中，Go 开发团队引入 Go Modules 构建模式。这个时候，GOPATH 构建模式与 Go Modules 构建模式各自独立工作，我们可以通过设置环境变量 GO111MODULE 的值在两种构建模式间切换。

然后，随着 Go 语言的逐步演进，从 Go 1.11 到 Go 1.16 版本，不同的 Go 版本在 GO111MODULE 为不同值的情况下，开启的构建模式几经变化，直到 Go 1.16 版本，Go Module 构建模式成为了默认模式。

所以，要分析 Go 各版本的具体构建模式的机制和切换，我们只需要找到这几个代表性的版本就好了。

我这里将 Go 1.13 版本之前、Go 1.13 版本以及 Go 1.16 版本，在 GO111MODULE 为不同值的情况下的行为做了一下对比，这样我们可以更好地理解不同版本下、不同构建模式下的行为特性，下面我们就来用表格形式做一下比对：

GO111MODULE	< Go 1.13	Go 1.13	Go 1.16
on	任何路径下都开启Go Module模式	任何路径下都开启Go Module模式	默认值：任何路径下都开启Go Module模式
auto	默认值：使用GOPATH 模式还是Go Module模式，取决于要构建的源码目录所在位置，以及是否包含go.mod文件。如果要构建的源码目录不在以GOPATH/src为根的目录体系下，且包含go.mod文件(两个条件缺一不可)，那么使用Go Module模式；否则使用传统的GOPATH模式	默认值：只要当前目录或父目录下有go.mod文件时，就开Go Module模式，无论源码目录是否在GOPATH外面	只有当前目录或父目录下有go.mod文件时，就开启Go Module模式，无论源码目录是否在GOPATH外面
off	GOPATH模式	GOPATH模式	GOPATH模式

了解了这些，你就能在工作中游刃有余的在各个 Go 版本间切换了，不用再担心切换后模式变化，导致构建失败了。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

在这一讲中，我们初步了解了 Go 语言构建模式的演化历史。

Go 语言最初发布时内置的构建模式为 GOPATH 构建模式。在这种构建模式下，所有构建都离不开 GOPATH 环境变量。在这个模式下，Go 编译器并没有关注依赖包的版本，开发者也无法控制第三方依赖的版本，导致开发者无法实现可重现的构建。

那么，为了支持可重现构建，Go 1.5 版本引入了 vendor 机制，开发者可以在项目目录下缓存项目的所有依赖，实现可重现构建。但 vendor 机制依旧不够完善，开发者还需要手工管理 vendor 下的依赖包，这就给开发者带来了不小的心智负担。

后来，Go 1.11 版本中，Go 核心团队推出了新一代构建模式：Go Module 以及一系列创新机制，包括语义导入版本机制、最小版本选择机制等。语义导入版本机制是 Go Moudle 其他机制的基础，它是通过在包导入路径中引入主版本号的方式，来区别同一个包的不兼

容版本。而且，Go 命令使用**最小版本选择**机制进行包依赖版本选择，这和当前主流编程语言，以及 Go 社区之前的包依赖管理工具使用的算法都有点不同。

此外，Go 命令还可以通过 GO111MODULE 环境变量进行 Go 构建模式的切换。但你要注意，从 Go 1.11 到 Go 1.16，不同的 Go 版本在 GO111MODULE 为不同值的情况下，开启的构建模式以及具体表现行为也几经变化，这里你重点看一下前面总结的表格。

现在，Go 核心团队已经考虑在后续版本中彻底移除 GOPATH 构建模式，Go Module 构建模式将成为 Go 语言唯一的标准构建模式。所以，**学完这一课之后，我建议你从现在开始就彻底抛弃 GOPATH 构建模式，全面使用 Go Module 构建模式**。

思考题

今天我们的思考题是：如何将基于 GOPATH 构建模式的现有项目迁移为使用 Go Module 构建模式？欢迎在留言区和我分享你的答案。

感谢你和我一起学习，也欢迎你把这节课分享给更多对 Go 构建模式感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 05 | 标准先行：Go项目的布局标准是什么？

精选留言 (16)

写留言

Aaron Liu
2021-10-25

看起来有点像node module，初学者从go module开始构建会比较好，理解一个方式

1 4

张诚
2021-10-25

目前网上讲包管理讲得最清晰最全面的内容了。

展开

作者回复: 过奖了:)

3

罗杰
2021-10-25

从构建来看，现在的确是掌握 Go 的好时机。之前入坑的时候，gopath 真的让人抓狂。

3

本

2021-10-25

if go version < 1.13: 项目移出GOPATH/src
go env -w GO111MODULE=on
go mod init module_path
go mod tidy

1

一步

2021-10-25

怎么指定自己开发 moduel 的版本呢？

展开

作者回复: 别急，07讲会给你答案。

1 1

Harold

2021-10-25

语义版本不能算go mod创新吧

展开

作者回复: 语义版本是独立的。但go module的"语义导入版本"机制的确与众不同。

1

2021-10-25

最近刚学go，然后被gopath搞得糊里糊涂，看了这一篇感觉学的时机恰到好处

1

2916012032@qq.com

2021-10-26

最小版本选择怎么兼容后续版本呢？

展开

liaomars

2021-10-26

把GOPATH构建下的项目目录复制一份出来，在这个目录下面开启 GO MODULES，执行：

go mod init && go mod tidy

展开

python玩家一枚

2021-10-26

老师您好，大概一个月前开始看go的书籍的，开始看的时候就构建项目纠结了很久，已经有的书籍基本上用的都是老方法gopath样式的pkg src bin 三件套，新的包括goland也是一个mod解决，然后又在一些书上看到说建议自己的项目按照github.com/xxxx/yyyy这种方式去建，还有又说最好是新版方式mod的也是放在gopath的src里面去玩，众说纷纭吧，这里有一些疑惑想请教下了，一般下载的三方依赖github.com/啥啥啥/这种也会被丢在gopath...

展开

jacky

2021-10-26

“这个例子中，C v1.3.0 是符合项目整体要求的版本集合中的版本最小的那个，于是 Go 命令选择了 C v1.3.0，而不是最新最大的 C v1.7.0。并且，Go 团队认为“最小版本选择”为 Go 程序实现持久的和可重现的构建提供了最佳的方案。” 这里最小的不是v1.1.0吗？怎么是v1.3.0呢

展开

lesserror

2021-10-25

感谢 Tony Bai 老师的分享，每篇文章我都认真拜读了。

本着持续精进的目的，每篇文章我都会提出一些自己的疑问，麻烦老师抽空解答一下。

1. 在$GOPATH模式下，寻找第三方包依赖的顺序是不是：先找 $GOROOT 然后找 $GOP...

展开

作者回复: 你的提问就是对专栏最好的支持，手工点赞！

下面回答你的问题：

1. 从gopath模式下，go build命令的输出来看，是这样的。
2. go.sum机制对于大多数开发人员都是透明的，属于高级话题。因此，在我的这个以入门和基础为重的专栏中并未深入展开。后>续我可能会在自己的博客或公号上谈谈go.sum机制，到时候，你可以去看看。

3. 相较于选择最新最大版本，选择最小版本出于几个考虑吧：

1) 对开发者而言，更易于理解和预测，就像课程中例子那样，我们根据依赖图可以很容易确定程序构建最终使用的依赖版本。
2) 对go核心团队来说，更容易实现，据说实现最小选择的代码也就几十行。
3) 更重要的是最小版本选择更容易实现可重现构建。试想一下，如果选择的是最大最新版本，那么针对同一份代码，其依赖包的最新最大版本在不同时刻可能是不同的，那么在不同时刻的构建，产生的最终文件就是不同的。

当然这一切的前提都是基于语义版本规范，对于不符合规范的module，相当于没有遵守契约，这套规则也就失效。这对任何语言来
说都是一样的。

4. 不属于。04讲说的那种情况是指在一个项目仓库中的不同目录下放置了多个go.mod，即一个项目中有多个module。

丶能
2021-10-25

提问！
“选择符合项目整体要求的最小版本”是可能选择，依赖最小版本与最新版本中任意版本，还是项目引入版本中的可选范围内的最小版本呢？

作者回复: 按照课程中的例子，A 要求 >=C v1.1.0，B 要求 C >=v1.3.0，那么选择同时满足A与B要求的最小版本，就是v1.3.0。如果选择v1.1.0则不满足A要求。

lawliet
2021-10-25

error while importing github.com/sirupsen/logrus: missing go.sum entry for module providing package golang.org/x/sys/unix (imported by github.com/sirupsen/logrus); to add:
go get github.com/sirupsen/logrus@v1.8.1

请问go mod tidy后还是会报错，但是能正常运行

展开

作者回复: 有全量命令执行与输出的日志么？

1

Bynow

2021-10-25

热乎，讲的真好。

展开

作者回复: 感谢支持:)

郑童文

2021-10-25

请问老师go.mod中module path的最后一段,必须和module 的名称一样吗？ 如果两者不同会出现什么情况？ 谢谢！

展开

作者回复: 严格来说，go.mod中 module后面的路径就是module的“名字”。但go官方一般不提module “名字”这个概念。module是一个go包集合，更多是一个命名空间的概念。go module唯一的标识就是module path。所以不存在你说的不一致情况。

1